

Scalable Partial Least Squares Regression on Grammar-Compressed Data Matrices

Yasuo Tabei
Japan Science and
Technology Agency, Japan
tabei.y.aa@m.titech.ac.jp

Yoshihiro Yamanishi
Kyushu University, Japan
yamanishi@bioreg.kyushu-
u.ac.jp

Hiroto Saigo
Kyushu University, Japan
saigo@inf.kyushu-u.ac.jp

Simon J. Puglisi
University of Helsinki, Finland
simon.puglisi@cs.helsinki.fi

ABSTRACT

With massive high-dimensional data now commonplace in research and industry, there is a strong and growing demand for more scalable computational techniques for data analysis and knowledge discovery. Key to turning these data into knowledge is the ability to learn statistical models with high interpretability. Current methods for learning statistical models either produce models that are not interpretable or have prohibitive computational costs when applied to massive data. In this paper we address this need by presenting a scalable algorithm for partial least squares regression (PLS), which we call compression-based PLS (cPLS), to learn predictive linear models with a high interpretability from massive high-dimensional data. We propose a novel grammar-compressed representation of data matrices that supports fast row and column access while the data matrix is in a compressed form. The original data matrix is grammar-compressed and then the linear model in PLS is learned on the compressed data matrix, which results in a significant reduction in working space, greatly improving scalability. We experimentally test cPLS on its ability to learn linear models for classification, regression and feature extraction with various massive high-dimensional data, and show that cPLS performs superiorly in terms of prediction accuracy, computational efficiency, and interpretability.

1. INTRODUCTION

Massive data are now abundant throughout research and industry, in areas such as biology, chemistry, economics, digital libraries and data management systems. In most of these fields, extracting meaningful knowledge from a vast amount of data is now the key challenge. For example, to remain competitive, e-commerce companies need to constantly analyze huge data of user reviews and purchasing histories [24].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

KDD '16, August 13-17, 2016, San Francisco, CA, USA

© 2016 ACM. ISBN 978-1-4503-4232-2/16/08...\$15.00

DOI: <http://dx.doi.org/10.1145/2939672.2939864>

In biology, detection of functional interactions of compounds and proteins is an important part in genomic drug discovery [29, 7] and requires analysis of a huge number of chemical compounds [3] and proteins coded in fully sequenced genomes [4]. There is thus a strong and growing demand for developing new, more powerful methods to make better use of massive data and to discover meaningful knowledge on a large scale.

Learning statistical models from data is an attractive approach for making use of massive high-dimensional data. However, due to high runtime and memory costs, learning of statistical models from massive data — especially models that have high interpretability — remains a challenge.

Partial least squares regression (PLS) is a linear statistical model with latent features behind high-dimensional data [26, 35, 36] that greedily finds the latent features by optimizing the objective function under the orthogonal constraint. PLS is suitable for data mining, because extracted latent features in PLS provide a low-dimensional feature representation of the original data, making it easier for practitioners to interpret the results. From a technical viewpoint, the optimization algorithm in PLS depends only on elementary matrix calculations of addition and multiplication. Thus, PLS is more attractive than other machine learning methods that are based on computationally burdensome mathematical programming and complex optimization solvers. In fact, PLS is the most common chemoinformatics method in pharmaceutical research.

However, applying PLS to massive high-dimensional data is problematic. While the memory for the optimization algorithm in PLS depends only on the size of the corresponding data matrix, storing all high-dimensional feature vectors in the data matrix consumes a huge amount of memory, which limits large-scale applications of PLS in practice. One can use *lossy compression* (e.g., PCA [14, 9] and *b-bit* minwise hashing [12, 20]) to compactly represent data matrices and then learn linear models on the compact data matrices [21]. However, although these lossy compression-based methods effectively reduce memory usage [21, 30], their drawback is that they cannot extract informative features from the learned models, because the original data matrices cannot be recovered from the compressed ones.

Grammar compression [2, 27, 16] is a method of *lossless compression* (i.e., the original data can be completely re-

Table 1: Summary of scalable learning methods of linear models.

	Approach	Compression Type	# of parameters	Interpretability	Optimization
PCA-SL [14, 9]	Orthogonal rotation	Lossy	2	Limited	Stable
bMH-SL [21]	Hashing	Lossy	3	Unable	Stable
SGD [33, 8]	Sampling	-	1	Limited	Unstable
cPLS (this study)	Grammar compression	Lossless	1	High	Stable

covered from grammar-compressed data) that also has a wide variety of applications in string processing, such as pattern matching [37], edit-distance computation [13], and q -gram mining [1]. Grammar compression builds a small context-free grammar that generates only the input data and is very effective at compressing sequences that contain many repeats. In addition, the set of grammar rules has a convenient representation as a forest of small binary trees, which enables us to implement various string operations without decompression. To date, grammar compression has been applied only to string (or sequence) data; however, as we will see, there remains high potential for application to other data representations. A fingerprint (or bit vector) is a powerful representation of natural language texts [23], bio-molecules [32], and images [11]. Grammar compression is expected to be effective for compressing a set of fingerprints as well, because fingerprints belonging to the same class share many identical features.

Contribution. In this paper, we present a new scalable learning algorithm for PLS, which we call *lossless compression-based PLS (cPLS)*, to learn highly-interpretable predictive linear models from massive high-dimensional data. A key idea is to convert high-dimensional data with fingerprint representations into a set of sequences and then build grammar rules for representing the sequences in order to compactly store data matrices in memory. To achieve this, we propose a novel grammar-compressed representation of a data matrix capable of supporting row and column access *while the data matrix is in a compressed format*. The original data matrix is grammar-compressed, and then a linear model is learned on the compressed data matrix, which allows us to significantly reduce working space. cPLS has the following desirable properties:

1. **Scalability:** cPLS is applicable to massive high-dimensional data.
2. **Prediction Accuracy:** cPLS can achieve high prediction accuracies for both classification and regression.
3. **Usability:** cPLS has only one hyper parameter, which enhances the usability of cPLS.
4. **Interpretability:** Unlike lossy compression-based methods, cPLS can extract features reflecting the correlation structure between data and class labels/response variables.

We experimentally test cPLS on its ability to learn linear models for classification, regression and feature extraction with various massive high-dimensional data, and show that cPLS performs superiorly in terms of prediction accuracy, computational efficiency, and interpretability.

2. LITERATURE REVIEW

Several efficient algorithms have been proposed for learning linear models on a large scale. We now briefly review the state of the art, which is also summarized in Table 1.

Principal component analysis (PCA) [14] is a widely used machine learning tool, and is a method of lossy compression, i.e., the original data cannot be recovered from compressed data. There have been many attempts to extend PCA [31, 28] and present a scalable PCA in distributed settings for analyzing big data [9]. For classification and regression tasks, a data matrix is compressed by PCA, and linear models are learned on the compressed data matrix by a *supervised learning method (SL)*, which is referred to as *PCA-SL*. Despite these attempts, PCA and its variants do not look at the correlation structure between data and output variables (i.e., class labels/response variables), which results in not only the inability of feature extractions in PCA but also the inaccurate predictions by PCA-SL.

Li et al. [21] proposed a compact representation of fingerprints for learning linear models by applying *b-bit minwise hashing (bMH)*. A d -dimensional fingerprint is conceptually equivalent to the set $s_i \subset \{1, \dots, d\}$ that contains element i if and only if the i -th bit in the fingerprint is 1. Li et al.’s method works as follows. We first pick h random permutations π_i , $i = 1, \dots, h$, each of which maps $[1, d]$ to $[1, d]$. We then apply a random permutation π on a set s_i , compute the minimum element as $\min(\pi(s_i))$, and take as a hash value its lowest b bits. Repeating this process h times generates h hash values of b bits each. Expanding these h values into a $(2^b \times h)$ -dimensional fingerprint with exactly h 1’s builds a compact representation of the original fingerprint.

Linear models are learned on the compact fingerprints by SL, which is referred to as *bMH-SL*. Although bMH-SL is applicable to large-scale learning of linear models, bMH is a method of lossy compression and cannot extract features from linear models learned by SL. Other hashing-based approaches have been proposed such as Count-Min sketch [5], Vowpal Wabbit [34], and Hash-SVM [25]. However, like bMH-SL, these algorithms cannot extract features, which is a serious problem in practical applications.

Stochastic gradient descent (SGD) [8, 33] is a computationally efficient algorithm for learning linear models on a large-scale. SGD samples ν feature vectors from an input dataset and computes the gradient vector from the sampled feature vectors. The weight vector in linear models is updated using the gradient vector and the learning rate μ , and this process is repeated until convergence. Unfortunately however, learning linear models using SGD is numerically unstable, resulting in low prediction accuracy. This is because SGD has three parameters (ν , μ , and C) that must be optimized if high classification accuracy is to be attained. Online learning is a specific version of SGD that loads an input dataset from the beginning and updates the weight vector in a linear model for each feature vector. *AdaGrad* [8] is an efficient online learning that automatically tunes parameters of ν and μ in SGD. Although online learning is space-efficient (owing to its online nature), it is also numerically unstable. Even worse, AdaGrad is applicable only

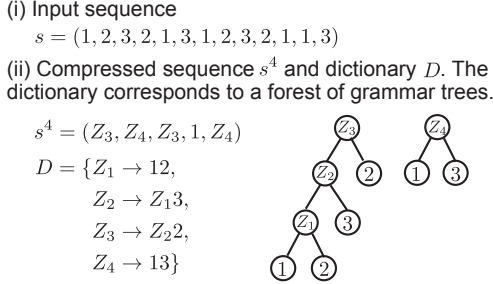


Figure 1: Illustration of grammar compression.

to differentiable loss functions, which limits its applicability to simple linear models, e.g., SVM and logistic regression, making the learned model difficult to interpret.

Despite the importance of scalable learning of interpretable linear models, no previous work has been able to achieve high prediction accuracy for classification/regression tasks and high interpretability of the learned models. We present a scalable learning algorithm that meets both these demands and is made possible by learning linear models on grammar-compressed data in the framework of PLS. Details of the proposed method are presented in the next section.

3. GRAMMAR COMPRESSION

Given a sequence of integers S , a *grammar-compressor* generates a context-free grammar (CFG) that generates S and only S . The grammar consists of a set of rules¹. Each rule is of the form $Z_i \rightarrow ab$. Symbols that appear on the left-hand side of any rule are called *non-terminals*. The remaining symbols are called *terminals*, all of which are present in the input sequence. Informally, a rule $Z_i \rightarrow ab$ indicates that on the way to recovering the original sequence from its grammar-compressed representation, occurrences of the symbol Z_i should be replaced by the symbol pair ab (the resulting sequence may then be subject to yet more replacements). A data structure storing a set of grammar rules is called a *dictionary* and is denoted by D . Given a non-terminal, the dictionary supports access to the symbol pair on the right-hand of the corresponding grammar rule, i.e., $D[Z_i]$ returns ab for rule $Z_i \rightarrow ab$. The original sequence can be recovered from the compressed sequence and D . The set of grammar rules in D can be represented as a forest of (possibly small) binary trees called *grammar trees*, where each node and its left/right children correspond to a grammar rule. See Figure 1 for an illustration.

The size of a grammar is measured as the number of rules plus the size of compressed sequence. The problem of finding the minimal grammar producing a given string is known to be NP-complete [2], but several approximation algorithms exist that produce grammars that are small in practice (see, e.g., [27, 19, 16]). Among these is the simple and elegant Re-Pair [19] algorithm, which we review next.

3.1 Re-Pair Algorithm

The Re-Pair grammar compression algorithm by Larsson and Moffat [19] builds a grammar by repeatedly replacing the most frequent symbol pair in an integer sequence with a

¹In this paper we assume without loss of generality that the grammar is in Chomsky Normal Form.

new non-terminal. Each iteration of the algorithm consists of the following two steps: (i) find the most frequent pair of symbols in the current sequence, and then (ii) replace the most frequent pair with a new non-terminal symbol, generating a new grammar rule and a new (and possibly much shorter) sequence. Steps (i) and (ii) are then applied to the new sequence and iterated until no pair of adjacent symbols appears twice.

Apart from the dictionary D that stores the rules as they are generated, Re-Pair maintains a hash table and a priority queue that together allow the most frequent pair to be found in each iteration. The hash table, denoted by H , holds the frequency of each pair of adjacent symbols ab in the current sequence, i.e., $H : ab \rightarrow \mathbb{N}$. The priority queue stores the symbol pairs keyed on frequency and allows the most frequent symbol pair to be found in step (i). In step (ii), a new grammar rule $Z_1 \rightarrow ab$ is generated where ab is the most frequent symbol pair and Z_1 is a new non-terminal not appearing in a sequence. The rule is stored in the dictionary D . Every occurrence of ab in the sequence is then replaced by Z_1 , generating a new, shorter sequence. This replacement will cause the frequency of some symbol pairs to change, so the hash table and priority queue are then suitably updated.

Let s^c denote a sequence generated at c -th iteration in the Re-Pair algorithm. For input sequence s in Figure 1, the most frequent pair of symbols is 12. Thus, we generate rule $Z_1 \rightarrow 12$ to be added to the dictionary D and replace all the occurrences of 12 by non-terminal Z_1 in s . After four iterations, the current sequence s^4 has no repeated pairs, and thus the algorithm stops. Dictionary D has four grammar rules that correspond to a forest of two small trees.

As described by Larsson and Moffat [19], Re-Pair can be implemented to run in linear time in the length of the input sequence, but it requires the use of several heavyweight data structures to track and replace symbol pairs. The overhead of these data structures (at least 128 bits per position) prevents the algorithm from being applied to long sequences, such as the large data matrices.

Another problem that arises when applying Re-Pair to long sequences is the memory required for storing the hash table: a considerable number of symbol pairs appear twice in a long sequence, and the hash table stores something for each of them, consuming large amounts of memory.

In the next section, we present scalable Re-Pair algorithms that achieve both space-efficiency and fast compression time on large data matrices. Specifically, our algorithms need only constant working space.

4. OUR GRAMMAR-COMPRESSED DATA MATRIX

Our goal is to obtain a compressed representation of a data matrix \mathbf{X} of n rows and d columns. Let x_i denote the i th row of the matrix represented as a fingerprint (i.e. binary vector). An alternative view of a row that will be useful to us is as a sequence of integers $s_i = (p_1, p_2, \dots, p_m)$, $p_1 < p_2 < \dots < p_m$, where $p_i \in s_i$ if and only if $x_i[p_i] = 1$. In other words the sequence s_i indicates the positions of the 1 bits in x_i .

In what follows we will deal with a differentially encoded form of s_i in which the difference for every pair of adjacent elements in s_i is stored, i.e., $s_i = (p_1, p_2, \dots, p_m)$ is encoded as $s_{gi} = (p_1, p_2 - p_1, p_3 - p_2, \dots, p_m - p_{m-1})$. This differ-

ential encoding tends to increase the number of repeated symbol pairs, which allows the sequences s_{gi} to be more effectively compressed by the Re-Pair algorithm. A grammar compressor captures the underlying correlation structure of data matrices: by building the same grammar rules for the same (sequences of) integers, it effectively compresses data matrices with many repeated integers.

4.1 Re-Pair Algorithms in Constant Space

We now present two ideas to make Re-Pair scalable without seriously deteriorating its compression performance. Our first idea is to modify the Re-Pair algorithm to identify top- k frequent symbol pairs in all rows s_{gi}^c in step (i) and replace all the occurrences of the top- k symbol pairs in all rows s_{gi}^c in step (ii), generating new k grammar rules and new rows s_{gi}^{c+1} . This new replacement process improves scalability by reducing the number of iterations required by roughly a factor of k .

Since we cannot replace both frequent symbol pairs ab and bc in triples abc in step (ii), we replace the first appearing symbol pair ab , preferentially. However, such preferential replacement can generate a replacement of a pair only once and can add redundant rules to a dictionary, adversely affecting compression performance. To overcome this problem, we replace the first and second appearances of each frequent pair at the same time and replace the next successive appearance of the frequent pair as usual, which guarantees generating grammar rules that appear at least twice.

Our second idea is to reduce the memory of the hash table by removing infrequent symbol pairs. Since our modified Re-Pair algorithm can work storing compressed sequences s_{gi}^c at each iteration c in a secondary storage device, the hash table consumes most of the memory in execution. Our modified Re-Pair generates grammar rules from only top- k frequent symbol pairs in the hash table, which means only frequent symbol pairs are expected to contribute to the compression. Thus, we remove infrequent symbol pairs from the hash table by leveraging the idea behind stream mining techniques originally proposed in [15, 6, 22] for finding frequent items in data stream. Our method is a counter-based algorithm that computes the frequency of each symbol pair and removes infrequent ones from the hash table at each interval in step (i). We present two Re-Pair algorithms using lossy counting and frequency counting for removing infrequent symbol pairs from the hash table. We shall refer to the Re-Pair algorithms using lossy counting and frequency counting as Lossy-Re-Pair and Freq-Re-Pair, respectively.

4.2 Lossy-Re-Pair

The basic idea of lossy counting is to divide a sequence of symbols into intervals of fixed length and keep symbol pairs in successive intervals in accordance with their appearance frequencies in a hash table. Thus, if a symbol pair has appeared h times in the previous intervals, it is going to be kept in the next h successive intervals.

Let us suppose a sequence of integers made by concatenating all rows s_{gi} of \mathbf{X} and let N be the length of the sequence. We divide the sequence into intervals of fixed-length ℓ . Thus, the number of intervals is N/ℓ . We use hash table H for counting the appearance frequency of each symbol pair in the sequence. If symbol pair ab has count $H(ab)$, it is ensured that ab is kept in hash table H until the next $H(ab)$ -th interval. If symbol pair ab first appears in

the q -th interval, $H(ab)$ is initialized as $qN/\ell + 1$, which ensures that ab is kept until at least the next interval, i.e., the $(qN/\ell + 1)$ -th interval. Algorithm 1 shows the pseudo-code of lossy counting.

The estimated number of symbol pairs in the hash table is $O(\ell)$ [22], resulting in $O(\ell \log \ell)$ bits consumed by the hash table.

Algorithm 1 Lossy counting. H : hash table, N : length of an input string at a time point, ℓ : length of each interval. Note that lossy counting can be used in step (i) in the Re-Pair algorithm.

```

1: Initialize  $N = 0$  and  $\Delta = 0$ 
2: function LOSSYCOUNTING( $a, b$ )
3:    $N = N + 1$ 
4:   if  $H(ab) \neq 0$  then
5:      $H(ab) = H(ab) + 1$ 
6:   else
7:      $H(ab) = \Delta + 1$ 
8:   if  $\lfloor \frac{N}{\ell} \rfloor \neq \Delta$  then
9:      $\Delta = \lfloor \frac{N}{\ell} \rfloor$ 
10:    for each symbol pair  $ab$  in  $H$  do
11:      if  $H(ab) < \Delta$  then
12:        Remove  $ab$  from  $H$ 
```

4.3 Freq-Re-Pair

The basic idea of frequency counting is to place a limit, v , on the maximum number of symbol pairs in hash table H and then keep only the most frequent v symbol pairs in H . Such frequently appearing symbol pairs are candidates to be replaced by new non-terminals, which generates a small number of rules.

The hash table counts the appearance frequency for each symbol pair in step (i) of the Re-Pair algorithm. When the number of symbol pairs in the hash table reaches v , Freq-Re-Pair removes the bottom ϵ percent of symbol pairs with respect to frequency. We call ϵ the vacancy rate. Algorithm 2 shows the pseudo-code of frequency counting. The space consumption of the hash table is $O(v \log v)$ bits.

Algorithm 2 Frequency counting. H : hash table, $|H|$: number of symbol pairs in H , v : the maximum number of symbol pairs in H , ϵ : vacancy rate. Note that frequency counting can be used in step (i) in the Re-Pair algorithm.

```

1: function FREQUENCYCOUNTING( $a, b$ )
2:   if  $H(ab) \neq 0$  then
3:      $H(ab) = H(ab) + 1$ 
4:   else
5:     if  $|H| \geq v$  then
6:       while  $v(1 - \epsilon/100) < |H|$  do
7:         for each symbol pair  $a'b'$  in  $H$  do
8:            $H(a'b') = H(a'b') - 1$ 
9:           if  $H(a'b') = 0$  then
10:            Remove  $a'b'$  from  $H$ 
11:    $H(ab) = 1$ 
```

5. DIRECT ACCESS TO ROW AND COLUMN

In this section, we present algorithms for directly accessing rows and columns of a grammar-compressed data matrix,

which is essential for us to be able to apply PLS on the compressed matrix in order to learn linear regression models.

5.1 Access to Row

Accessing the i -th row corresponds to recovering the original s_i from grammar-compressed s_{gi}^c . We compute this operation by traversing the grammar trees. For recovering the i -th row s_i , we start traversing the grammar tree having a node of the q -th symbol $s_{gi}^c[q]$ as a root for each q from 1 to $|s_{gi}^c|$. Leaves encountered in the traversal must have integers in sequence s_{gi} , which allows us to recover s_{gi} via tree traversals, starting from the nodes with non-terminal $s_{gi}^c[q]$ for each $q \in [1, |s_{gi}^c|]$. We recover the original i -th row s_i from s_{gi} by cumulatively adding integers in s_{gi} from 1 to $|s_{gi}|$, i.e., $s_i[1] = s_{gi}[1]$, $s_i[2] = s_{gi}[2] + s_i[1], \dots, s_i[|s_{gi}|] = s_{gi}[|s_{gi}|] + s_i[|s_{gi}| - 1]$.

5.2 Access to Column

Accessing the j -th column of a grammar-compressed data matrix requires us to obtain a set of row identifiers R such that $x_{ij} = 1$ for $i \in [1, n]$, i.e., $R = \{i \in [1, n]; x_{ij} = 1\}$. This operation enables us to compute the transpose \mathbf{X}^T from \mathbf{X} in compressed format, which is used in the optimization algorithm of PLS.

$P[Z_i]$ stores a summation of terminal symbols as integers at the leaves under the node corresponding to terminal symbol Z_i in a grammar tree. For example, in Figure 1, $P[Z_1] = 3$, $P[Z_2] = 6$, $P[Z_3] = 8$ and $P[Z_4] = 4$. P can be implemented as an array that is randomly accessed from a given non-terminal symbol. We shall refer to P as the weight array. The size of P depends only on the grammar size.

The j -th column is accessed to check whether or not $x_{ij} = 1$ in compressed sequence s_{gi}^c , for each $i \in [1, n]$. We efficiently solve this problem on grammar-compressed data matrix by using the weight array P . Let u_q store the summation of weights from the first symbol $s_{gi}^c[1]$ to the q -th symbol $s_{gi}^c[q]$, i.e., $u_q = P[s_{gi}^c[1]] + P[s_{gi}^c[2]] + \dots + P[s_{gi}^c[q]]$, and let $u_0 = 0$. If u_q is not less than j , the grammar tree with the node corresponding to a symbol $s_{gi}^c[q]$ as a root can encode j at a leaf. Thus, we traverse the tree in depth-first order from the node corresponding to symbol $s_{gi}^c[q]$ as follows. Suppose $Z = s_{gi}^c[q]$ and $u = u_{q-1}$. Let Z_ℓ (resp. Z_r) be a (resp. b) of $Z \rightarrow ab$ in D . (i) if $j < u$, we go down to the left child in the tree; (ii) otherwise, i.e., $j \geq u$, we add $P[Z_\ell]$ to u and go down to the right child. We continue the traversal until we reach a leaf. If $s = j$ at a leaf, this should be $x_{ij} = 1$ at row i ; thus we add i to solution set R . Algorithm 3 shows the pseudo-code for column access.

6. CPLS

In this section we present our cPLS algorithm for learning PLS on grammar-compressed data matrices. We first review the PLS algorithm on uncompressed data matrices. NIPALS [35] is the conventional algorithm for learning PLS and requires the deflation of the data matrix involved. We thus present a non-deflation PLS algorithm for learning PLS on compressed data matrices.

6.1 NIPALS

Let us assume a collection of n data samples and their output variables $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ where $y_i \in \mathbb{R}$. The output variables are assumed to be centralized as $\sum_{i=1}^n y_i =$

Algorithm 3 Access to the j -th column on grammar-compressed data matrix. R : solution set of row identifiers i at column j s.t. $x_{ij} = 1$.

```

1: function ACCESSCOLUMN( $j$ )
2:   for  $i$  in  $1..n$  do
3:      $u_0 = 0$ 
4:     for  $q$  in  $1..|s_{gi}^c|$  do
5:        $u_q = u_{q-1} + P[s_{gi}^c[q]]$ 
6:       if  $j \leq u_q$  then
7:         RECURSION( $i, j, s_{gi}^c[q], u_{q-1}$ )
8:       break
9:   function RECURSION( $i, j, Z, u$ )
10:  if  $Z$  is a terminal symbol then
11:    if  $u + Z = j$  then
12:      Add  $i$  to  $R$ 
13:    return
14:  Set  $Z_\ell$  (resp.  $Z_r$ ) as  $a$  (resp.  $b$ ) of  $Z \rightarrow ab$  in  $D$ 
15:  if  $u + P[Z_\ell] > j$  then
16:    RECURSION( $i, j, Z_\ell, u$ )  $\triangleright$  Go to left child
17:  else
18:    RECURSION( $i, j, Z_r, u + P[Z_\ell]$ )  $\triangleright$  Go to right child

```

0. Denote by $y \in \mathbb{R}^n$ the vector of all the training output variables, i.e., $y = (y_1, y_2, \dots, y_n)^T$.

The regression function of PLS is represented by the following special form,

$$f(x) = \sum_{i=1}^m \alpha_i w_i^T x,$$

where the w_i are weight vectors reducing the dimensionality of x ; they satisfy the following orthogonality condition:

$$w_i^T \mathbf{X}^T \mathbf{X} w_j = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}. \quad (1)$$

We have two kinds of variables w_i and α_i to be optimized. Denote by $\mathbf{W} \in \mathbb{R}^{d \times m}$ the weight matrix i -th column of which is weight vector w_i , i.e., $\mathbf{W} = (w_1, w_2, \dots, w_m)$. Let $\alpha \in \mathbb{R}^m$ be a vector whose i -th element is α_i , i.e., $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_m)^T$. Typically, \mathbf{W} is first optimized and then α is determined by minimizing the least squares error without regularization,

$$\min_{\alpha} \|y - \mathbf{X}\mathbf{W}\alpha\|_2^2. \quad (2)$$

By computing the derivative of equation (2) with respect to α and setting it to zero, α is obtained as follows:

$$\alpha = (\mathbf{W}^T \mathbf{X}^T \mathbf{X} \mathbf{W})^{-1} \mathbf{W}^T \mathbf{X}^T y. \quad (3)$$

The weight vectors are determined by the following greedy algorithm. The first vector w_1 is obtained by maximizing the squared covariance between the mapped feature $\mathbf{X}w$ and the output variable y as follows: $w_1 = \operatorname{argmax}_w \operatorname{cov}^2(\mathbf{X}w, y)$, subject to $w^T \mathbf{X}^T \mathbf{X} w = 1$, where $\operatorname{cov}(\mathbf{X}w, y) = y^T \mathbf{X}w$. The problem can be analytically solved as $w_1 = \mathbf{X}^T y$.

For the i -th weight vector, the same optimization problem is solved with additional constraints to maintain orthogonality,

$$w_i = \operatorname{argmax}_w \operatorname{cov}^2(\mathbf{X}w, y), \quad (4)$$

subject to $w^T \mathbf{X}^T \mathbf{X} w = 1$, $w^T \mathbf{X}^T \mathbf{X} w_j = 0$, $j = 1, \dots, i-1$. The optimal solution of this problem cannot be obtained analytically, but NIPALS solves it indirectly. Let us define the

i -th latent vector as $t_i = \mathbf{X}w_i$. The optimal latent vectors t_i are obtained first and the corresponding w_i is obtained later. NIPALS performs the deflation of design matrix \mathbf{X} to ensure the orthogonality between latent components t_i as follows, $\mathbf{X} = \mathbf{X} - t_i t_i^T \mathbf{X}$. Then, the optimal solution has the form, $w_i = \mathbf{X}^T y$.

Due to the deflation, $\mathbf{X} = \mathbf{X} - t_i t_i^T \mathbf{X}$, NIPALS completely destroys the structure of \mathbf{X} . Thus, it cannot be used for learning PLS on grammar-compressed data matrices.

6.2 cPLS Algorithm

We present a non-deflation PLS algorithm for learning PLS on grammar-compressed data matrices. Our main idea here is to avoid deflation by leveraging the connection between NIPALS [35] and the Lanczos method [18] which was originally proposed for recursive fitting of residuals without changing the structure of a data matrix.

We define residual $r_{i+1} = (r_i - (y^T t_{i-1}) t_{i-1})$ that is initialized as $r_1 = y$. The i -th weight vector is updated as $w_i = \mathbf{X}^T (r_{i-1} - (y^T t_{i-1}) t_{i-1})$, which means w_i can be computed without deflating the original data matrix \mathbf{X} . The i -th latent vector is computed as $t_i = \mathbf{X}w_i$ and is orthogonalized by applying the Gram-Schmidt orthogonalization to the i -th latent vector t_i and previous latent vectors t_1, t_2, \dots, t_{i-1} as follows, $t_i = (\mathbf{I} - \mathbf{T}_{i-1} \mathbf{T}_{i-1}^T) \mathbf{X}w_i$, where $\mathbf{T}_{i-1} = (t_1, t_2, \dots, t_{i-1}) \in \mathbb{R}^{n \times (i-1)}$. The non-deflation PLS algorithm updates the residual r_i instead of deflating \mathbf{X} , thus enabling us to learn PLS on grammar-compressed data matrices.

cPLS is the non-deflation PLS algorithm that learns PLS on grammar-compressed data matrices. The input data matrix is grammar-compressed and then the PLS is learned on the compressed data matrix by the non-deflation PLS algorithm. Our grammar-compressed data matrix supports row and column accesses directly on the compressed format for computing matrix calculations of addition and multiplication, which enables us to learn PLS by using the non-deflation PLS algorithm. Let \mathbf{X}_G be the grammar-compressed data matrix of \mathbf{X} . Algorithm 4 shows the pseudo-code of cPLS. Since our grammar-compression is lossless, the cPLS algorithm on grammar-compressed data matrices learns the same model as the non-deflation PLS algorithm on uncompressed data matrices and so achieves the same prediction accuracy.

Algorithm 4 The cPLS algorithm. \mathbf{X}_G : the grammar-compressed data matrix of \mathbf{X} .

```

1:  $r_1 = y$ 
2: for  $i = 1, \dots, m$  do
3:    $w_i = \mathbf{X}_G^T r_i$  ▷ access to column
4:   if  $i = 1$  then
5:      $t_1 = \mathbf{X}_G w_i$  ▷ access to row
6:   else
7:      $t_i = (\mathbf{I} - \mathbf{T}_{i-1} \mathbf{T}_{i-1}^T) \mathbf{X}_G w_i$  ▷ access to row
8:      $t_i = t_i / \|t_i\|_2$ 
9:      $r_{i+1} = r_i - (y^T t_i) t_i$ 
10: Compute the coefficients  $\alpha$  using equation (3).
```

We perform feature extraction after line 3 at each iteration in Algorithm 4. The features corresponding to the top- u largest weights w_i are extracted. Due to the orthogonality condition (1), the extracted features give users a novel insight for analyzing data, which is shown in Section 7.

The cPLS algorithm has three kinds of variables to be

optimized: w_i , r_i , and t_i . The memory for w_m is $O(md)$ and the memory for t_m and r_i is $O(mn)$. Thus, the total memory for the variables in cPLS is $O(m \min(n, d))$ highly depending on parameter m . The parameter m controls the amount of fitting of the model to the training data and is typically chosen to optimize the cross validation error. Since the cPLS algorithm learns the model parameters efficiently, m can be set to a small value, which results in overall space-efficiency.

7. EXPERIMENTS

In this section, we demonstrate the effectiveness of cPLS with massive datasets. We used five datasets, as shown in Table 2. "Book-review" consists of 12,886,488 book reviews in English from Amazon [24]. We eliminated stop-words from the reviews and then represented them as 9,253,464 dimensional fingerprints, where each dimension of the fingerprint represents the presence or absence of a word. "Compound" is a dataset of 42,682 chemical compounds that are represented as labeled graphs. We enumerated all the subgraphs of at most 10 vertices from the chemical graphs by using gSpan [38] and then converted each chemical graph into a 52,099 dimensional fingerprint, where each dimension of the fingerprint represents the presence or absence of a chemical substructure. "Webspam" is a dataset of 16,609,143 fingerprints of 350,000 dimensions². "CP-interaction" is a dataset of 216,121,626 compound-protein pairs, where each compound-protein pair is represented as a 3,621,623 dimensional fingerprint and 300,202 compound-protein pairs are interacting pairs according to the STITCH database [17]. We used the above four datasets for testing the binary classification ability. "CP-intensity" consists of 1,329,100 compound-protein pairs represented as 682,475 dimensional fingerprints, where the information about compound-protein interaction intensity was obtained from several chemical databases (e.g., ChEMBL, BindingDB and PDSP Ki). The intensity was observed by IC50 (half maximal (50%) inhibitory concentration). We used the "CP-intensity" dataset for testing the regression ability. The number of all the nonzero dimensions in each dataset is summarized in the #nonzero column in Table 2, and the size for storing fingerprints in memory by using 32bits for each element is written in the memory column in Table 2. We implemented all the methods by C++ and performed all the experiments on one core of a quad-core Intel Xeon CPU E5-2680 (2.8GHz). We stopped the execution of each method if it had not finished within 24hours in the experiments. In the experiments, cPLS did not use a secondary storage device for compression, i.e., cPLS compressed data matrices by loading all data in memory.

7.1 Compression Ability and Scalability

First, we investigated the influence on compression performance of the top- k parameter in our Re-Pair algorithms. For this setting, we used the Lossy-Re-Pair algorithm, where parameter ℓ is set to the total length of all rows in an input data matrix in order to keep all the symbols in the hash table. We examined $k = \{1 \times 10^4, 2.5 \times 10^4, 5 \times 10^4, 7.5 \times 10^4, 10 \times 10^4\}$ for the Book-review, Compound and Webspam datasets and examined $k = \{1 \times 10^5, 2.5 \times 10^5, 5 \times 10^5, 7.5 \times 10^5, 10 \times 10^5\}$ for the CP-interaction and CP-intensity datasets.

²The dataset is downloadable from <http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html>.

Table 2: Summary of datasets.

Dataset	Label type	Number	Dimension	#nonzeros	Memory (mega bytes)
Book-review	binary	12,886,488	9,253,464	698,794,696	2,665
Compound	binary	42,682	52,099,292	914,667,811	3,489
Webspam	binary	350,000	16,609,143	1,304,697,446	4,977
CP-interaction	binary	216,121,626	3,621,623	32,831,736,508	125,243
CP-intensity	real	1,329,100	682,475	28,865,055,991	110,111

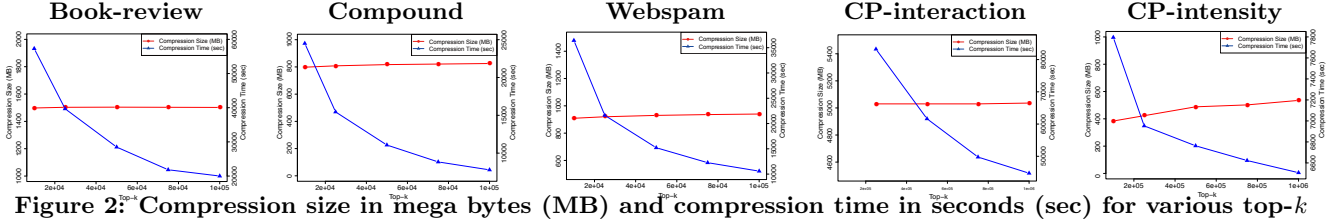


Figure 2: Compression size in mega bytes (MB) and compression time in seconds (sec) for various top- k

Figure 2 shows compression size and compression time for various top- k . We observed a trade-off between compressed size and compression time for all the datasets. The smaller the compressed size, the larger the compression time for larger values of k . In particular, significantly faster compression time was possible at the cost of only slightly worse compression. For example, Lossy-Re-Pair took 57,290 seconds to compress the Book-review dataset and its size was 1,498 mega bytes (MB) for $k=10000$. When $k=100000$, compression time dropped to 20,004 seconds (less than half), while compressed size increased negligibly to 1,502MB.

The same trends for the Book-review dataset were observed in the other datasets, which suggests that in practice a large value of k can be chosen for fast compression, without adversely affecting compression performance. Notably, we observed our compression method to be particularly effective for the larger datasets: CP-interaction and CP-intensity. The original sizes of CP-interaction and CP-intensity were 125GB and 110GB, respectively, while the compressed sizes of CP-interaction and CP-intensity were at most 5GB and at 535MB, respectively. Our compression method thus achieved compression rates of 4% and less than 1% for CP-interaction and CP-intensity, respectively. Such significant reductions in data size enable the PLS algorithm to scale to massive data.

Next, we evaluated the performance of Lossy-Re-Pair and Freq-Re-Pair, where parameters $\ell=\{1\text{MB}, 10\text{MB}, 100\text{MB}, 1000\text{MB}\}$ were examined for Lossy-Re-Pair, and parameters $v=\{1\text{MB}, 10\text{MB}, 100\text{MB}, 1000\text{MB}\}$ and $\epsilon=\{30\}$ were examined for Freq-Re-Pair. Table 3 shows the compressed size, compression time and the working space used for the hash table in Lossy-Re-Pair and Freq-Re-Pair. We observed that both Lossy-Re-Pair and Freq-Re-Pair achieved high compression rates using small working space. Such efficiency is crucial when the goal is to compress huge data matrices that exceed the size of RAM; our Re-Pair algorithm can compress data matrices stored in external memory (disk). For compressing the CP-interaction dataset, Lossy-Re-Pair and Freq-Re-Pair consumed 16GB and 13GB, respectively, achieving a compressed size of 5GB. We observed the same tendency for the other datasets (See Table 3).

7.2 Prediction Accuracy

We evaluated the classification and regression capabilities

of cPLS, PCA-SL, bMH-SL and SGD. Following the previous works [39, 21], we randomly selected 20% of samples for testing and used the remaining 80% of samples for training. cPLS has one parameter m , so we selected the best parameter value among $m = \{10, 20, \dots, 100\}$ that achieved the highest accuracy for each dataset. The PCA phase of PCA-SL has one parameter deciding the number of principal components m , which was chosen from $m = \{10, 25, 50, 75, 100\}$ whose maximum value of 100 is the same as that of cPLS's parameter m . Linear models were learned with LIBLINEAR [10], one of the most efficient implementations of linear classifiers, on PCA's compact feature vectors, where the hinge loss of linear SVM for classification and the squared error loss for regression were used with L_2 -regularization. The learning process of PCA-SL [14, 9] has one parameter C for L_2 -regularization, which was chosen from $C = \{10^{-5}, 10^{-4}, \dots, 10^5\}$. For PCA-SL [14, 9], we examined all possible combinations of two parameters (m and C) and selected the best combination achieving the highest accuracy for each dataset. The hashing process of bMH-SL [21] has two parameters (the number of hashing values h and the length of bits b), so we examined all possible combinations of $h = \{10, 30, 100\}$ and $b = \{8, 16\}$. As in PCA-SL, linear models were learned with LIBLINEAR [10] on bMH's compact feature vectors, where the hinge loss of linear SVM for classification and the squared error loss for regression were used with L_2 -regularization. The learning process of bMH-SL [21] has one parameter C for L_2 -regularization, which was chosen from $C = \{10^{-5}, 10^{-4}, \dots, 10^5\}$. For bMH-SL, we examined all possible combinations of three parameters (h , b , and C) and selected the best combination achieving the highest accuracy for each dataset. We implemented SGD on the basis of the AdaGrad algorithm [8] using the logistic loss for classification and the squared error loss for regression with L_2 -regularization. SGD [8] has one parameter C for L_2 -regularization, which was also chosen from $C = \{10^{-5}, 10^{-4}, \dots, 10^5\}$. We measured the prediction accuracy by the area under the ROC curve (AUC) for classification and Pearson correlation coefficient (PCC) for regression. Note that AUC and PCC return 1 for perfect inference in classification/regression, while AUC returns 0.5 for random inference and PCC returns 0 for random inference. We report the best test accuracy under the above experimental settings for each method below.

Table 3: Compression size in mega bytes (MB), compression time in seconds (sec), and working space for hash table (MB) for varying parameter ℓ in Lossy-Re-Pair and v in Freq-Re-Pair for each dataset.

Book-review								
	Lossy-RePair ℓ (MB)				Freq-RePair v (MB)			
	1	10	100	1000	1	10	100	1000
compression size (MB)	1836	1685	1502	1501	2021	1816	1680	1501
compression time (sec)	9904	12654	19125	20004	2956	2355	3165	21256
working space (MB)	1113	1931	7988	8603	292	616	3856	6724
Compound								
	Lossy-RePair ℓ (MB)				Freq-RePair v (MB)			
	1	10	100	1000	1	10	100	1000
compression size (MB)	1288	859	825	825	1523	1302	825	825
compression time (sec)	5096	7053	7787	7946	1362	1587	8111	8207
working space (MB)	1113	1926	5030	5030	292	616	3535	3535
Webspam								
	Lossy-RePair ℓ (MB)				Freq-RePair v (MB)			
	1	10	100	1000	1	10	100	1000
compression size (MB)	1427	948	940	940	2328	2089	1050	940
compression time (sec)	6953	10585	10584	10964	2125	2799	7712	11519
working space (MB)	1112	1923	7075	7075	292	616	3856	5539
CP-interaction								
	Lossy-RePair ℓ (MB)				Freq-RePair v (MB)			
	10	100	1000	10000	10	100	1000	10000
compression size (MB)	-	5199	5139	5036	20307	9529	5136	5136
compression time (sec)	24hours	55919	44853	43756	24565	39647	47230	48653
working space (MB)	-	9914	16650	16635	616	3856	13796	13796
CP-intensity								
	Lossy-RePair ℓ (MB)				Freq-RePair v (MB)			
	10	100	1000	10000	10	100	1000	10000
compression size (MB)	558	543	540	535	588	535	535	535
compression time (sec)	8103	6479	6494	6657	5423	5848	5859	5923
working space (MB)	1936	3552	3722	3738	616	2477	2477	2477

Table 4: Results of cPLS, PCA-SL, bMH-SL and SGD for various datasets. Dspace: the working space for storing data matrix (MB), Ospace: the working space for optimization algorithm and Ltime: learning time (sec).

cPLS					
Data	m	Dspace(MB)	Ospace(MB)	Ltime(sec)	AUC/PCC
Book-review	100	1288	15082	21628	0.96
Compound	20	786	7955	1089	0.83
Webspam	60	890	7736	4171	0.99
CP-interaction	40	4367	53885	35880	0.77
CP-intensity	60	472	10683	33969	0.67
PCA-SL					
Data	m/C	Dspace(MB)	Ospace(MB)	Ltime(sec)	AUC/PCC
Book-review	100/1	14747	110	6820	0.70
Compound	25/0.1	12	1	6	0.65
Webspam	50/1	200	2	129	0.99
CP-interaction	-	-	-	>24hours	-
CP-intensity	100/0.1	1521	11	42	0.11
bMH-SL					
Data	$b/h/C$	Dspace(MB)	Ospace(MB)	Ltime(sec)	AUC/PCC
Book-review	100/16/0.01	2457	110	1033	0.95
Compound	30/16/10	2	1	1	0.62
Webspam	30/16/10	20	2	2	0.99
CP-interaction	30/16/0.1	12366	1854	10054	0.77
CP-intensity	100/16/0.1	253	11	45	0.54
SGD					
Data	C	Dspace(MB)	Ospace(MB)	Ltime(sec)	AUC/PCC
Book-review	10	-	1694	57	0.96
Compound	10	-	9539	83	0.82
Webspam	10	-	3041	85	0.99
CP-interaction	1	-	663	3163	0.75
CP-intensity	0.1	-	124	280	0.04

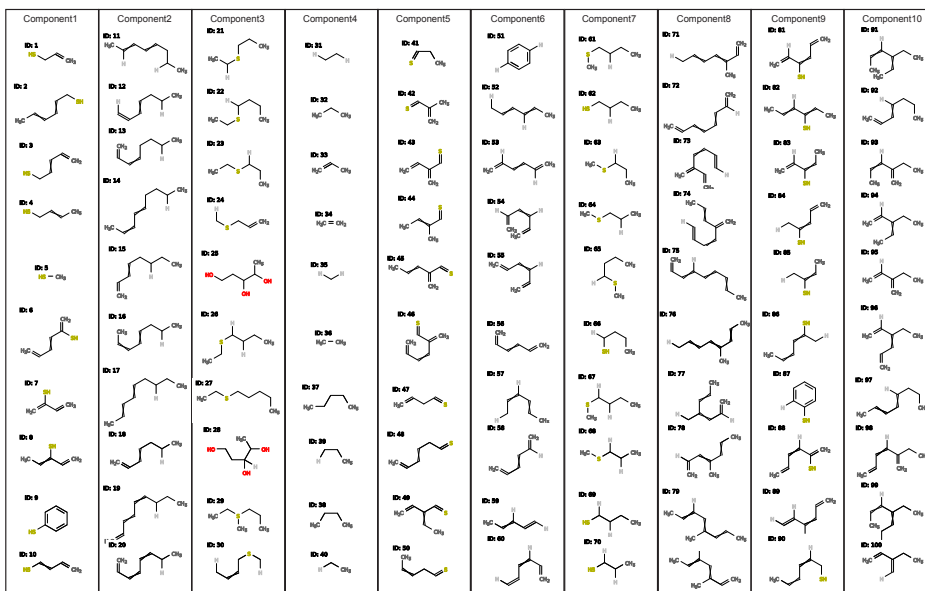


Figure 3: Extracted features for the top 10 latent components in the application of cPLS to the Compound dataset. Each column represents the highly weighted features (chemical substructures) of a latent component.

Table 4 shows the prediction accuracy, working space and training time of cPLS, PCA-SL, bMH-SL, and SGD. The working space for the storing data matrix and the working space needed for optimizations were separately evaluated. While PCA-SL and bMH-SL significantly reduced the working space for storing data matrices, the classification and regression accuracies were low. Since PCA-SL and bMH-SL compress data matrices without looking at the correlation structure between data and output variables for compressing data matrices, high classification and regression accuracies were difficult to achieve.

SGD significantly reduced the working space, since it did not store data matrices in memory. Classification and regression accuracies of SGD were not high, because of the instability of the optimization algorithm. In addition, SGD is applicable only to simple linear models, making the learned model difficult to interpret.

Our proposed cPLS outperformed the other methods (PCA-SL, bMH-SL, and SGD) in terms of AUC and PCC and significantly reduced the working space. The results showed cPLS’s efficiency for learning PLS on compressed data matrices while looking at the correlation structure between data and output variables. Such a useful property enables us to extract informative features from the learned model.

7.3 Interpretability

Figure 3 shows the top-10 highly-weighted features that were extracted for each component in the application of cPLS to the Compound dataset, where one feature corresponds to a compound chemical substructure. It was observed that structurally similar chemical substructures were extracted together as important features in the same component, and the extracted chemical substructures differed between components. This observation corresponds to a unique property of cPLS. Analysing large-scale compound structure data is of importance in pharmaceutical applications, especially for rational drug design. For example, the extracted chemical substructures are beneficial for users who

want to identify important chemical fragments involved in therapeutic drug activities or adverse drug reactions.

8. CONCLUSIONS AND FUTURE WORK

We presented a scalable algorithm for learning interpretable linear models — called cPLS — which is applicable to large-scale regression and classification tasks. Our method has the following appealing properties:

1. **Scalability:** cPLS is applicable to large numbers of high-dimensional fingerprints (see Sections 7.1 and 7.2).
2. **Prediction Accuracy:** The optimization of cPLS is numerically stable, which enables us to achieve high prediction accuracies (see Section 7.2).
3. **Usability:** cPLS has only one hyperparameter to be tuned in cross-validation experiments (see Section 6.2).
4. **Interpretability:** Unlike lossy compression-based methods, cPLS can extract informative features reflecting the correlation structure between data and class labels (or response variables), which makes the learned models easily interpretable (see Section 7.3).

In this study, we applied our proposed grammar compression algorithm to scaling up PLS, but in principle it can be used for scaling up other machine learning methods or data mining techniques. An important direction for future work is therefore the development of scalable learning methods and data mining techniques based on grammar-compression techniques. Such extensions will open the door for machine learning and data mining methods to be applied in various large-scale data problems in research and industry.

9. ACKNOWLEDGMENTS

This work was supported by MEXT/JSPS Kakenhi (24700140, 25700004 and 25700029), the JST PRESTO program, the Program to Disseminate Tenure Tracking System, MEXT and Kyushu University Interdisciplinary Programs in Education and Projects in Research Development, and the Academy of Finland via grant 294143.

10. REFERENCES

- [1] P. Bille, P. H. Cording, and I. L. Gortz. Compact q-gram profiling of compressed strings. In *CPM*, pages 62–73, 2013.
- [2] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51:2554–2576, 2005.
- [3] B. Chen, D. Wild, and R. Guha. PubChem as a source of polypharmacology. *JCIM*, 49:2044–2055, 2009.
- [4] The Uniprot Consortium. The universal protein resource (uniprot) in 2010. *NAR*, 38:D142–D148, 2010.
- [5] G. Cormode and Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55:58–75, 2005.
- [6] D. Demaine, A. López-Ortiz, and I. Munro. Frequency estimation of internet packet streams with limited space. In *ESA*, pages 348–360, 2002.
- [7] C.M. Dobson. Chemical space and biology. *Nature*, 432(7019):824–828, 2004.
- [8] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12:2121–2159, 2011.
- [9] T. Elgamal, M. Yabandeh, A. Abounaga, W. Mustafa, and M. Hefeeda. sPCA: Scalable principal component analysis for big data on distributed platforms. In *SIGMOD*, 2015.
- [10] R. Fan, K. W. Chang, C. J. Hsieh, X. R. Wang, and C. J. Lin. LIBLINEAR: A library for large linear classification. *JMLR*, pages 1871–1874, 2008.
- [11] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, 2002.
- [12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.
- [13] D. Hermelin, D. H. Landau, and O. Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In *STACS*, pages 529–540, 2009.
- [14] I. T. Jolliffe. *Principal Component Analysis*. Springer, 1986.
- [15] R. Karp, S. Shenker, and C. Papadimitriou. A simple algorithm for finding frequent elements in sets and bags. *TODS*, 28:51–55, 2003.
- [16] J. C. Kieffer, E. Yang, G. J. Nelson, and P. C. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46(4):1227–1245, 2000.
- [17] M. Kuhn, D. Szklarczyk, A. Franceschini, M. Campillos, C. von Mering, L.J. Jensen, A. Beyer, and P. Bork. STITCH 2: An interaction network database for small molecules and proteins. *NAR*, 38(suppl 1):D552–D556, 2010.
- [18] C. Lanczos. An iteration method for the solution of the eigen value problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45:255–282, 1950.
- [19] J. Larsson and A. Moffat. Offline dictionary-based compression. In *DCC*, pages 296–305, 1999.
- [20] P. Li and A. C. König. b-bit minwise hashing. In *WWW*, pages 671–680, 2010.
- [21] P. Li, A. Shrivastava, J. L. Moore, and A. C. König. Hashing algorithms for large-scale learning. In *NIPS*, pages 2672–2680, 2011.
- [22] G. Manku and R. Motwani. Approximate frequency counts over data stream. In *VLDB*, volume 5, pages 346–357, 2002.
- [23] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.
- [24] J. McAuley and J. Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *RecSys*, 2013.
- [25] Y. Mu, G. Hua, W. Fan, and S. Chang. Hash-SVM: Scalable kernel machines for large-scale visual classification. In *CVPR*, pages 979–986, 2014.
- [26] R. Rosipal and N. Krämer. Overview and recent advances in partial least squares. In *Subspace, Latent Structure and Feature Selection*, LNCS, pages 34–51. Springer, 2006.
- [27] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *TCS*, 302(1–3):211–222, 2003.
- [28] B. Schölkopf, A. Smola, and K. R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299–1319, 1998.
- [29] B. Stockwell. Chemical genetics: Ligand-based discovery of gene function. *Nature Reviews Genetics*, 1:116–125, 2000.
- [30] Y. Tabei and Y. Yamanishi. Scalable prediction of compound-protein interactions using minwise-hashing. *BMC Systems Biology*, 7:S3, 2013.
- [31] M. E. Tipping and C. M. Bishop. Mixtures of probabilistic principal component analysers. *Neural Computation*, 11:443–482, 1999.
- [32] R. Todeschini and V. Consonni. *Handbook of Molecular Descriptors*. Wiley-VCH, 2002.
- [33] Y. Tsuruoka, J. Tsujii, and S. Ananiadou. Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty. In *ACL and AFNLP*, pages 477–485, 2009.
- [34] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *ICML*, pages 1113–1120, 2009.
- [35] H. Wold. Path models with latent variables: The NIPALS approach. In *Quantitative Sociology: International Perspectives on Mathematical and Statistical Model Building*, pages 307–357. Academic Press, 1975.
- [36] S. Wold, M. Sjöström, and L. Eriksson. PLS-regression: a basic tool of chemometrics. *Chemometrics and Intelligent Laboratory Systems*, 58:109–130, 2001.
- [37] T. Yamamoto, H. Bannai, S. Inenaga, and M. Takeda. Faster subsequence and don’t-care pattern matching on compressed texts. In *CPM*, pages 309–322, 2011.
- [38] X. Yan and J. Han. gSpan: graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.
- [39] H.F. Yu, C.J. Hsieh, K.W. Chang, and C.J. Lin. Large linear classification when data cannot fit in memory. In *KDD*, pages 833–842, 2010.